Towards Discovery, Specification, and Verification of Component Usage

Chang Liu, Debra J. Richardson

Department of Information and Computer Science
University of California, Irvine, Irvine, CA 92697

{liu, djr}@ics.uci.edu

Technical Report UCI-ICS-99-22

May 14, 1999

Abstract

Additional problems of software testing arise when applications under test are developed in the component-based approach. Component misuse is one of them. The component misuse problem occurs when a component is used in a way that differs from the component producer's expectation. This paper explores the cause of the component misuse problem and proposes a technique to discover, specify, and verify component usage. This technique utilizes regular expressions as the formalism to deal with component usage. This research is part of the software retrospector effort, which aims at a better testing solution for component-based software.

1. Introduction

The component-based software development approach [Szy98] brings us many benefits, including lower development cost, shorter time-to-market, higher quality, etc. However, these benefits come with a price, including increased difficulty in component production, application composition, and both component and application testing.

It is more difficult to produce a reusable component than a non-reusable one, because the component writer must think about all potential future use of the component, not just the most immediate use. For example, Boehm recognized that the cost of producing a reusable software component is 50% higher than its non-reusable counterpart if the reusable component is to be reused across multiple product lines [BS92].

It is also more difficult to compose an application from off-the-shelf components than from custom-built ones, because the interface of a COTS component is fixed when it reaches component users. DeLine found that component packaging mismatch is a significant problem [Del99]. Batory and Geraci found that component composition is a challenge and needs special design rules [BG97].

Testing component-based software has many pitfalls not typical of traditionally developed applications. Weyuker identified several complications of component-based software testing, including testing "without source code and access to the personnel and expertise used to create the component" [Wey98]. There are several reasons: COTS components typically are delivered without source code. How can application builders be sure to use the components correctly? COTS components should have been rigorously tested before delivery, yet the application testers have no way of knowing if they have and to what degree. How can they assure the quality of COTS components? Using traditional black-box testing (or even white-box testing, if by chance COTS components' source code are available) loses some potential productivity gains of employing high-quality COTS components. Why should they be tested again? COTS components may have been rigorously tested but are not shipped with test code. If application testers want to take advantage of already-tested COTS components, where can they find information about what has been tested and what has not and how can they reuse that information in testing the application?

The goal of the component-based approach is to reuse everything that has been done for a component, namely design, coding, debugging, and testing effort. Yet when a COTS component is delivered, typically what arrives is binary object code and a few undefined documents. Design and testing documents are typically not delivered with the component and thus cannot be reused.

Component usage is another piece of information that is missed in delivery. Component writers design a component according to certain component usage assumptions. Correct usage is critical to successful composition of the component with the rest of the application, yet, expected component usage is never explicitly specified. It may be expressed in the form of source code comments, component interface comments, and sample programs that use the component. It is difficult, however, to understand the expected usage of a component expressed in these ways unless it is a very simple component. Thus, misuse of reusable components happens often. Currently, the main means of detecting component misuse is through testing and debugging, which is very

expensive. The difficulty of detecting misuse of reusable components is another obstacle to the component-based approach. We need a better solution to make the component-based approach more appealing.

This paper discusses the component misuse problem and presents a potential solution. In Section 2, we describe in detail the component misuse problem. An example is used to illustrate the problem. Our solution to this problem, which is a technique for discovery, specification, and verification of component usage, is presented in Section 3.

Our work on component usage is a part of the software retrospector research effort [LR98]. Software retrospectors are software entities that reside in software components and store testing-related information to assist software testing of component-based application software. The information stored in a retrospector includes component usage information, test execution history and results, and test coverage. The component usage specification described in this paper is the part of a software component retrospector that describes valid and/or unacceptable usage information.

2. The Component Misuse Problem

The component misuse problem occurs when a component is used in a way that is different from what the component producer expects. Component misuse does not necessarily always end up with application failures. However, it does affect the clarity of application architecture and frequently does cause application failures.

We use a list traverse example to illustrate the component misuse problem. Suppose we have a guarded linked list component. A guarded linked list always has a guard at the beginning of the list so that the value of the list pointer is never NULL. It has a group of functions to provide the traverse capability to its users. Let us borrow C++ syntax to describe the interface of the traverse functions of this component. The interface description is in Figure 1.

```
Class GuardedLinkedList {
  ...
  Bool init();
  Bool next();
  Item * getCurrent();
} T;
```

Figure 1. Interface of the component GuardedLinkedList

Just based on function names, a component user can guess some usage information from this interface definition: "init()" is used to prepare for a traverse; "next()" is used to traverse to the next item; "getCurrent()" is used to get the current item for further processing. It is not clear, however, exactly how these functions should be used together. One possible implementation is to use "init()" to move an internal traverse pointer to the first item in the logical list (not the guard) so that users can use "getCurrent()" to access it. Later, "next()" can be used to move the pointer to the next item so that users can use "getCurrent()" to get access to it. A possible usage of such a list is shown in Figure 2.

```
void traverse()
{
  Item * p;
  if (T.init() == TRUE)
    do
    {
      ...
      p = T.getCurrent();
      ... // process the item pointed by "p"
    } until (T.next()==FALSE);
}
```

Figure 2. One possible usage of GuardedLinkedList

While the above implementation is intuitive, there could be others. Another possible implementation is to use "init()" to move the internal traverse pointer to the guard instead of the first logical item in the linked list. In this case, the "next()" should be called before users can use "getCurrent()" to access the first item. An example of using the traverse component under this assumption is shown in Figure 3.

```
Void traverse()
{
  Item * p;
  if (T.init() == TRUE)
    while (T.next()==TRUE)
    {
      ...
      p = T.getCurrent();
      ... // process the item pointed by "p"
    }
}
```

Figure 3. Another possible usage of GuardedLinkedList

Clearly, while the program shown in Figure 2 is correct under the first usage assumption, it will suffer the component misuse problem if the implementation uses the second usage assumption. And vice versa for the program shown in Figure 3.

Now let us look at a real life example with a similar component. YACL (Yet Another Class Library) is a class library in C++ [Sri96], which has an iterator class that provides traverse functionality for other classes. The interface of the iterator class is shown in Figure 4. Just like the hypothetical example discussed above, the usage assumption is not clear from this interface specification, and the author did not document expected usage. We can figure out one acceptable usage pattern, however, by looking at the piece of sample code shown in Figure 5. This sample code, embedded in comments in an unrelated YACL source file, shows a usage of the iterator. It explains that `Reset(), More()` are designed to be conveniently used in the FOR statement of the C++ language. Still it is unclear whether or not the program in Figure 6 is correct. The only way to figure this out is by looking at the component source code (which is at best expensive and at worst impossible since the code is not usually available in the component-based approach) or by experimenting with the component (which is even more expensive and unreliable). Even worse, if the component user simply assumes the wrong usage and is not aware of other possible usage assumptions, component misuse can only be detected by testing and debugging (and then only by chance).

```
Template <class T>
Class YACL_BASE CL_Iterator: public CL_Object {
Public:
    ~CL_Iterator() {};
    virtual void      Reset () = 0;
    // Reset the iterator to the beginning. (Pure virtual.)
    virtual bool      More  () = 0;
    // Return TRUE if there are more elements to be returned.
    // (Pure virtual.)
    virtual const T&  Next  () = 0;
    // Return the next element.  (Pure virtual.)
};
```

Figure 4. An iterator class (Exact excerpt from YACL 1.6)

```
//          CL_StringIntMapIterator iter (myMap);
//          CL_StringIntAssoc assoc;
//          for (iter.Reset(); iter.More(); ) {
//              assoc = iter.Next();
//              // Process the pair "assoc" here....
//          }
```

Figure 5. Comments that explains usage assumption about the iterator (Exact excerpt from YACL 1.6)

```
CL_StringIntMapIterator iter (myMap);
CL_StringIntAssoc assoc;
for (iter.Reset(); iter.More(); ) {
    process1(iter.Next());
    process2(iter.Next());
    process3(iter.Next());
}
```

Figure 6. Another possible usage of the iterator

The iterator class is simple. It has only three functions. In a more complex situation with more functions and more subtle relationships among those functions, it is necessarily more difficult to discover and correct misuse of a component.

YACL is just a class library. It does not utilize any advanced component technology. However, none of the component technologies we have now solve the component misuse problem. We still rely on software testing and debugging to locate misuse in component-based approach.

At the heart of the component misuse problem is the inability for a component provider to explicitly specify component usage and the absence of a formal mechanism to guarantee the correct usage of a reusable component. The more internal states and the more interdependencies among interface functions a component has, the more difficult it is to use the component and to locate and correct component misuses by testing and debugging.

In summary, the component misuse problem can occur even in simple examples. The component provider assumes certain relationships among component functions.

These relationships are not explicitly advertised in the interface signature specification. Additional documentation in the forms of sample programs or informal descriptions are the usual ways to implicitly and informally communicate these assumptions to component users. The availability and accuracy of these communications, however, are not guaranteed.

3. Discover, Specify, and Verify Component Usage

We propose to solve the component misuse problem by explicitly defining component usage in a usage specification language and then verifying correct usage. We choose regular expressions [ASU86] as the underlying formalism to specify component usage because we feel that the expected usage of a component can be exactly described as a string over an alphabet that consists of the functions in the interface of that component.

We do not want to burden component providers by making usage specification mandatory. Instead, we first try to discover component usage from sample programs provided with that component. Section 3.1 has the description of our component usage discovery technique.

In the mean time, we want to allow and encourage component providers to explicitly describe valid and/or unacceptable component usage. This is because the component usage discovery algorithm can not discover all valid usage. It also does not have the ability to tell what is a typical usage since it does not have the knowledge of the meaning of an example. Nor is it possible to discover unacceptable usage from examples. Section 3.2 describes in detail the language we propose for component usage specification.

With the component usage specification from either derived usage discovery or explicit usage specification, we are now equipped to attack the component misuse problem. We propose both static and dynamic verification techniques to detect component misuse. Component usage verification is described in Section 3.3.

3.1 Component Usage Discovery

During the development of reusable components, component producers inevitably write sample code to either experiment with and/or test their components. These test driver-like sample programs embody important usage assumptions. If we are able to discover component usage from this sample code, we can get component usage information without requiring any extra effort from component producers.

We intend to obtain component usage information "for free" by analyzing sample code that uses a component and producing a component usage specification as regular expressions. The regular expressions are defined over an alphabet that consists of the interface functions of the component under analysis. For the example component in Figure 1, the alphabet is {*init, next, getCurrent*}. If a function name is overloaded, each signature is treated as a different member of the alphabet.

To be able to focus on semantically related functions, we introduce the notion of function groups. A function group is a subset of the interface of a component. Functions in a function group are related in certain aspects but are unrelated to other functions with regard to those particular aspects. By definition, a component could have more than one function group. Component producers can define a list of related functions as a function

group before component usage discovery so that the discovery algorithm can focus only on these functions and produce more meaningful and more readable usage specifications. For a given component, component producers could define more than one function group. Function groups are not disjoint. The default function group consists of all functions in the component interface. For each function group, a regular expression will be discovered as an acceptable usage specification.

The component usage algorithm is shown in Figure 7. For the sake of algorithm description, let us assume C++ class-like mechanisms are used as to construct components. The usage discovery algorithm takes as input the component under analysis *C*, a function group *FG*, and a sample program *P* that uses *C*. Assume *P* has only one procedure and contains no GOTO statements. *USAGE* refers to the usage of function group *FG* of component *C*.

Let *T* be the set of all variables in *P*.

Let *DEF*(*t*,*FG*) be the set of all *simple* statements[1] in *P* that use one or more functions in *FG* to assign a value for variable $t \in T$.

For each variable t ∈ T and each *definition* d ∈ *DEF*(t,C), a regular expression defining usage within the sample program *P* will be discovered. Each statement *s* reached by the *definition* d is examined in syntactic orders (i.e., textual order in the source code), where *d reaches s* if there is a path from the statement immediately following *d* to *s* such that *d* is not redefined [ASU86]. A regular expression *RE* is used to record the discovery as the algorithm proceeds.

With these assumptions, the algorithm in Figure 7 can discover component usage and record the result in *USAGE*.

[1] A simple statement is anything other than a compound statement such as a block, branch, or loop.

USAGE = ∅
for each variable $t \in T$ and for each *definition* $d \in DEF$(t,C) **loop**
RE = the use at *d*

For example, if the use is "*t*.`func1()`", the function (letter) "`func1`" is concatenated to RE. Let us describe this as *RE* = *RE* `func1`. A use "*t*.`func1()` + *t*.`func2()`" results in *RE* = *RE* `func1 func2`.

for each statement *s* reached by *d* **loop**

- if *s* is a simple statement that assigns a value to *t* (it is the end of the *reaching definition d*) **exit loop.**
- if *s* is a simple statement that does not use *t* or functions in *FG*, do nothing.
- if *s* is a simple statement that uses *t* and functions in *FG*, concatenate this use to *RE* (as in the step taken at the *definition d* above).
- if *s* is a branch statement, assume without loss of generality that the statement is
  if (Expression) { Left } else { Right }.
  Suppose using this same algorithm, the regular expression discovered from Expression, Left, and Right with regard to *d* are *RE.Expression*, *RE.Left*, and *RE.Right* respectively, then the branch statement results in *RE* = *RE* *RE*.Expression (*RE*.Left | *RE*.Right).
- if *s* is a loop statement, let us assume without loss of generosity that the statement is
  do { Body } until (Expression)
  Suppose using this same algorithm, the regular expression discovered from Body and Expression with regard to *d* are *RE.Body* and *RE.Expression*, then the loop statement results in *RE* = *RE* (*RE*.Body *RE*.Expression)+.

end loop
USAGE = *USAGE* ∪ *RE*
end loop

Figure 7. Component usage discovery algorithm

This component usage discovery algorithm can discover obvious component usage from straightforward sample code. If applied on the sample code in Figure 2, it produces this regular expression: `"init (getCurrent next)+"`. For Figure 3, the result is `"init next (getCurrent next)*"`.

There are a number of shortcomings of this algorithm:

First, the complexity of this algorithm is exponential in the size of the sample code under analysis.

Second, GOTO statements, which will make the scope of *def* points hard to control, are not allowed.

Third, there are cases in which the algorithm is not able to discover the usage because it lacks the ability to resolve pointer aliases. For example, for the pieces of code in both Figure 8 and Figure 9, the ideal usage specification discovered should be "f1 f2",

but {f1, f2} will be discovered instead. In Figure 8, {f1, f2} is discovered because for variable "p", the *def* point at line 1 generates "f1", while the *def* point at line 3 generates "f2". In Figure 9, {f1, f2} is discovered because "f1" is discovered for variable "p" and "f2" is discovered for variable "pp". In general, the algorithm does not expect variables that hold reference to the component under analysis to change value.

```
1  p = pp = a pointer to the component under analysis;
2  p->f1();
3  p = pp;
4  p->f2();
```

Figure 8. An example of problematic code

```
1  p = a pointer to the component under analysis;
2  p->f1();
3  pp=p;
4  pp->f2();
```

Figure 9. Another example of problematic code

Fourth, the algorithm assumes that in the sample program, all related uses of component under analysis are located within one procedure or function. This is a serious limitation. Future support for multi-procedure programs is possible. For example, an inlining technique might be able to handle some inter-procedural discovery, although recursion may be problematic.

Despite these problems, we feel that many sample programs are straightforward and within the effective range of this algorithm. For complex usage that can not be automatically discovered from sample code, the component producers can explicitly specify component usage, as described in the next section.

3.2 Component Usage Specification

Component producers best know valid component usage. With a little extra effort, the usage of a component can be described in the form of regular expressions.

Component producers can specify three types of usage, as shown in Figure 10:

- The first type is desirable usage, which is recommended usage by component producers. This is the best way to use the component because it is designed this way.
- The second type is acceptable usage, which describes all legitimate usage of a component. Any usage other than this may cause unexpected result.
- The third type is typical incorrect usage. Although any usage not recognized by the "acceptable" regular expression is wrong, it is beneficial for component producers to be able to define "typical incorrect" usage and specifically point out common mistakes to warn future component users. It is not necessary to specify all unacceptable usages.

```
UsageOf CL_Iterator Is
  Desirable: (Reset (More Next)* More)* ;
  Acceptable: (Reset (More+ Next)* [More])*;
  Typical Incorrect: (Reset Next (More [Next])*)+;
End
```

Figure 10. Usage specification of component "CL_Iterator" (underlined words are keywords)

The usage specification in Figure 10 basically says that the usage exemplified in Figure 5 is the most desirable usage. It is acceptable, however, if a component user calls "More()" repeatly before calling "Next()". A common error that the component producer wants users to avoid is using "Next()" right after "Reset()", as described by the regular expression after keyword "Typical Incorrect".

Function groups are defined implicitly by each individual regular expression. By definition, functions that do not appear in a regular expression are not related to the group of functions that are used in the regular expression and thus their usage is not restricted by this particular regular expression. For this reason, component producers may specify a list of regular expressions instead of a single regular expression to describe any of the three usage types.

The usage specification in Figure 10 can be embedded into source code so that it is easier to keep the usage specification up-to-date. This is shown in Figure 11. If the component is to be delivered without source code, the usage specification should be extracted from source code and delivered as a separate document. In cases when separate component specifications are maintained and delivered, usage specification can be embedded in component specifications, too.

The syntax of the component usage specification language is described in Figure 12.

```
//UsageOf CL_Iterator Is
//  Desirable: Reset (More Next)* More
//  Acceptable: Reset (More+ Next)* More
//  Typical Incorrect: Reset (Next More)*
//End
template <class T>
class YACL_BASE CL_Iterator: public CL_Object {
public:
    ~CL_Iterator() {};
    virtual void      Reset () = 0;
    // Reset the iterator to the beginning. (Pure virtual.)
    virtual bool      More  () = 0;
    // Return TRUE if there are more elements to be returned.
    // (Pure virtual.)
    virtual const T&  Next  () = 0;
    // Return the next element.  (Pure virtual.)
};
```

Figure 11. Component usage specification can be embedded into source code as comments

```
Usage-spec :
"UsageOf" component-name "Is"
[ "Desirable" regexp-list ]
[ "Acceptable" regexp-list ]
[ "Typical" "Unacceptable" regexp-list ]
"End"
;

component-name:
NAME
;

regexp-list :
      // empty
|
      regexp-list regexp ";"
;

regexp :
      letter
|
      regexp +          // zero or more occurrence
|
      regexp *          // one or more occurrence
|
      ( regexp )
|
      [ regexp ]        // optional
|
      regexp | regexp   // or
|
      regexp regexp
;

letter :
      func-name
|
      "\"" func-name "\""
|
      "\"" func-name "(" arg-list ")" "\""
         // so that overloaded functions can be distinguished
;

func-name :
      NAME
;
```

Figure 12. Syntax of the component usage specification language

3.3 Component Usage Verification

Component usage verification works with a component usage specification to automatically verify the usage of application software that uses that component.

There are two ways to verify component usage. First, static verification using the usage discovery algorithm presented in Section 3.1 is the most obvious approach. This works when the application under verification uses the component in a straightforward way and does not have structures that the algorithm can not handle (such as those pointed out in Section 3.1). Using the static approach, we only have to check if the usage discovered can be recognized by any regular expression in the specification and inform

component users of the result. If it is recognized by a "Desirable" regular expression, it is a good use. If an "Acceptable" regular expression but not a "Desirable" one recognizes it, it is a legitimate use. If it is recognized by a "Typical Incorrect" regular expression, it is a use that the component producer specifically designated as unacceptable and told component users to avoid.

For the component `GuardedLinkedList`, shown in Figure 1, if the specification in Figure 13 specifies its usage, the static component usage verification algorithm can find out that the program in Figure 2 is wrong and the program in Figure 3 is correct.

```
UsageOf GuardedLinkedList Is
  Desirable: (init (next getCurrent)* next)*;
  Acceptable: (init (next [getCurrent])* )*;
  Typical Incorrect: (init getCurrent (next [getCurrent])* )+;
End
```

Figure 13. Usage specification of component "CL_Iterator" (underlined words are keywords)

Static verification verifies component usage only by program analysis. No test execution is needed. Static verification is precise given the specification.

In cases when the usage discovery algorithm is not able to discover usage from the application under analysis, dynamic usage verification is necessary. Dynamic usage verification is similar to software testing; the application is executed with test data. The difference is, when performing dynamic usage verification, a wrapper of the component is used to capture all component function calls. The function call sequence is then verified against the component usage specification.

If the program in Figure 2 is tested by test inputs such that the statements inside the loop are only executed once, the function call sequence captured by the wrapper of component `GuardedLinkedList` is "`(init getCurrent next)`". This will be recognized by the "Typical Incorrect" regular expression in the usage specification in Figure 13, thus detecting a wrong use of the component `GuardedLinkedList`.

Dynamic usage verification works with any application using a component and is a complement to the static verification approach. This approach, however, is imprecise; sophisticated test data selection is critical to the effectiveness and efficiency of dynamic usage verification.

When checking a discovered usage or a captured function call sequence against a usage specification, incomplete sentences are always allowed except for "Typical Incorrect" usage. This is because a program could terminate at any time, and thereby terminate interaction with a component at any time.

4. Future Work, Related Work, and Summary

We have not yet fully implemented the ideas presented in this paper. The most important future work is to implement the usage discovery tool, the usage specification parser, and the usage verification analyzer.

We would also like to experiment with more sophisticated components where more subtle interdependencies among interface functions are critical to the successful use

of the components. Generally, a component with more internal states, such as network socket communication components, has more interdependencies among interface functions. We would like to find such real components from publicly available sources to demonstrate that component usage specification works better than sample code or documentation, especially in complicated cases.

For components with overloaded functions, the usage specification might be a little long since all the argument types have to be included as part of the letter in the alphabet. Macro mechanism can be included in the specification language to make it more readable in this situation.

Many problems arise when the component-based approach becomes a major development method. The difficulty of expressing, understanding, and verifying component usage are among them. We propose to use regular expressions to explicitly specify component usage, to use program analysis technique to discover and statically verify component usage, and to use test execution of the wrapped component with the application to dynamically verify component usage. The specification we developed for the iterator example shows that usage specification can clearly communicate subtle usage requirements. However, we must still develop supporting technology and study more complicated examples to show the practicality of this approach.

A number of researchers are working to attack the problems of component-based software. DeLine tried to use "flexible packaging" to avoid the "packaging mismatch" problem [Del99]. While this technique helps fitting different types of component together, it does not help guarantee that components are used in an expected way. Batory and Geraci developed a technique to validate transformational component composition using "design rule checking" [BG97]. There are also other researchers working on component-based software testing. Rosenblum, for instance, is exploring test adequacy of component-based software [Ros97].

Our approach to component-based software testing problem is to allow software components to carry testing and usage information with them so that testing activities are more informed and therefore more effective. The mechanism that we designed for this purpose is called retrospectors [LR98]. The work on component usage presented here is a part of the retrospector effort.

References

[ASU86] A. Aho, R. Sethi, J. Ullman, Compilers – Principles, Techniques, and Tools, Addison-Wesley, 1986

[BG97] D. Batory and B.J. Geraci. "Composition Validation and Subjectivity in GenVoca Generators", IEEE Transactions on Software Engineering, 23(2):67-82, February 1997.

[BS92] B.W. Boehm and W.L. Scherlis. "Megaprogramming", in Proceedings of the DARPA Software Technology Conference 1992, pp. 63-82, Los Angeles, CA, April 1992.

[CPRZ89] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria", IEEE Transactions on Software Engineering, November 1989.

[Del99] Robert DeLine, "Avoiding packaging mismatch with Flexible Packaging." To appear in Proceedings of the International Conference on Software Engineering, Los Angeles, California, 1999.

[LR98] Chang Liu, Debra J. Richardson, "Software Components with Retrospectors", International Workshop on the Role of Software Architecture in Testing and Analysis, Marsala, Sicily, Italy, July 1998.

[Ros97] D. Rosenblum, Adequate Testing of Component-Based Software. Technical Report, UCI-ICS-97-34, University of California, Irvine. August, 1997

[Szy98] C. Szyperski, Component Software - Beyond Object-Oriented Programming, Addison-Wesley. 1998

[Sri96] M. A. Sridhar, Building Portable C++ Applications with YACL, Addison-Wesley, 1996

[Wey98] Elaine J. Weyuker, "Testing Component-Based Software: A Cautionary Tale", IEEE Software Volume 15 Number 5, September/October 1998.